DEFENDANT'S
EXHIBIT
DX-724
2:15-cv-00011-RSP

AspectCOOL: An experiment in design and implementation of aspect-oriented language

Enis Avdičaušević, Mitja Lenič, Marjan Mernik, Viljem Žumer
University of Maribor
Faculty of Electrical Engineering and Computer Science
{enis, mitja.lenic, marjan.mernik, zumer}@uni-mb.si

Abstract

Aspect-oriented programming (AOP) is a promising technique helping programmers to easier reason about, develop and maintain programs. AOP improves reusability since components with a clearly defined functionality, which is not tangled with different aspects, are much easier to reuse. In order to explore different AOP concepts a general-purpose aspect-oriented language AspectCOOL has been designed and implemented. Among the different concepts, which we explored, the most important is the separate compilation of aspect and component code. Using this concept aspects can be applied on already compiled components, which improves their reusability.

Keywords: aspect-oriented programming, programming language design and implementation, separate compilation, reuse

1 Introduction

The separation of concerns is a major abstraction technique which helps us to manage software complexity. The crucial problem in software development is to find a suitable system decomposition, which allows clear separation of concerns and modular design. However, some properties of software systems: for example exception handling or synchronization, cannot be captured into a single module. These properties crosscut the basic functionality of the system. This breaks modular structure of the system, which is essential for efficient software reuse and maintenance. AOP [1] extends already existing paradigms, such as functional programming and object-oriented programming with concept of aspect. In aspect-oriented languages two different programming concepts can be identified: components and aspects. Components are used to capture the basic functionality of the system, resulting from its functional decomposition. These components have clearly defined structure and responsibilities. Aspects implement additional properties of the system, which influence basic functionality of the system. With aspects it is possible to capture such properties in a modular way. Programs, which use aspects to describe these properties, are much easier to understand, maintain and reuse.

Components are described using component language, which is usually some general-purpose language, for example, Java or Lisp. Aspects are described using a special language called aspect language. Aspect languages can be general-purpose languages, like AspectJ [2] or they can be designed for the special application domain. In order to achieve desired properties of the system, we need an aspect weaver, which combines both component and aspect programs into the final program. With aspect program, the programmer describes special segments of program code (in AspectJ it is called *advice*), which is applied (inserted) on special places in the component program called *join points*. Appearance of join points in component program is not same in all aspect-oriented environments. It depends, for example, on the component language and purpose (domain) of the aspect language. In object-oriented languages, for example, join point can be represented as method call. In that case it is possible to describe additional actions in aspect language, which can be executed before or after method execution. Aspect weaver combines component and aspect language by inserting advices on appropriate join points. There are also several possible approaches to weaving. In this paper a design and implementation of a general-purpose aspect-oriented language AspectCOOL is presented. AOP improves reusability since components with a clearly defined functionality, which is not tangled with different aspects, are much easier to reuse. However, reuse is

still limited since many aspect weavers require a source code of components and aspects to compose a final program. The main feature of AspectCOOL is the separate compilation of aspect and component code where aspects can be applied on already compiled components, which further improves their reusability.
The organization of this paper is as follows. In section 2 motivation and related work are described. Design and implementation of AspectCOOL language is described in section 3. Our approach to aspect weaving is presented in section 4, followed by conclusion in section 5.

2 Motivation and Related Work

Our research interests are the design and implementation of programming languages. We have been involved for a long time in implementation of compiler generators based on attribute grammars [3], [4]. More recently, we have also become interested in the implementation of domain-specific languages. Aspect-oriented languages can be observed as domain-specific languages, which provide language mechanisms that explicitly capture crosscutting structure. Crosscutting concerns are expressed with either domain-specific aspect languages, or general-purpose aspect language. We have designed and implemented the general-purpose aspect language AspectCOOL as an extension of the class based object-oriented language COOL[1] to explore different aspect-oriented programming concepts and ideas such as: separation of advices and join points, explicit join points, caller to callee method call transitions as join points, separate compilation of aspect and component code, and applying aspects to components where the source code is not available. Some of these ideas has already been proposed in the literature.
AspectJ is the most widely known general-purpose aspect-oriented language [6]. It is a practical and simple extension of Java with the aim of empirical assessment of AOP and to support the AOP community. In AspectJ, aspect weaving is mostly done at compile-time to avoid unnecessary run-time overhead and to find as many programming errors as possible at compile-time. AspectJ is not just a pre-processor but rather a full compiler. However, current implementation has the following limitations [6]: javac is used as back-end instead of generating class files directly, separate compilation is not supported; access to all source code is required and a full recompilation is performed whenever any part of the source code is changed. Our approach enables separate compilation and has no such limitations.
In [7] author suggests that the aspect description (aspect class) should be divided into two parts where advices and join points are separately defined. Using this approach the advice part is potentionally much more reusable. Unfortunately, this was just a proposition and implementation has not yet begun.
AOP in BETA programming language is supported by a fragment system [8][9]. The fragment system handles the physical organization of programs and enables separate compilation, separation of interfaces and implementations, information hiding, program variants, inter-module dependencies, etc. In the fragment system the join points, named slots, are explicit. Moreover, they can be at the attribute, operation and statement levels. To support the static type system in BETA, the compiler makes a static analysis of the aspect code to ensure it is statically correct. Hence, aspect cannot be imposed on different classes, which is a limitation of the approach. With our approach there is no such limitation. Separate compilation and static semantic conformance were also achieved also at non-explicit join points.
The benefits of applying aspects to Java Commercial Off-the-Shelf (COTS) software is described in [10]. Since the source code of COTS software is not available or the compiled classes may be unavailable before they are loaded into the Java Virtual Machine, the aspects have to be applied to already compiled classes. In [10] class loading is intercepted and the class bytecode is rewritten before the class is instantiated by the Java Virtual Machine. In our approach bytecode rewriting is not used since we have full access to the component compiler and instead of class loading interception the method calls are intercepted.

[1] do not confuse this general-purpose language with aspect language COOL developed by C. Lopes as a part of her Ph.D. thesis [5]

3 COOL and AspectCOOL

In this section the component language COOL (Classroom Object-Oriented Language) and the aspect language AspectCOOL are presented. The most important goal in the development of the AspectCOOL language was to explore the possibilities for separate compilation of components and aspects.
The program consists of components and the aspect part of the program. Components are described as classes in the component language. The aspect part is described in the aspect language and consists of aspect classes and the application part. Aspect classes contain advices, which can be applied on join points. In the application part, advices are applied on concrete join points.
The languages COOL and AspectCOOL were developed using a formal method named multiple attribute grammar inheritance [4]. This formal method is supported by our tool LISA ver. 2.0. With this tool we had the chance to develop our languages in modular and incremental way.

3.1 Component language

COOL is an object-oriented language used for studying object-oriented language design and implementation. The basic language COOL had to be upgraded in order to prepare foundations for separate compilation. COOL has the following features:

- Dynamic loading - Modules must be compiled separately. Each component is compiled in a separate object file. Loading of modules is done on demand.
- Method call interception - To support AOP we must be able to intercept method calls with all parameters from the caller and the callee.
- Easy environment transportation - Local environments inside methods should be easily transported to the aspects and accessible outside the local method environment. This is necessary for the transportation of actual parameters to aspects.
- Reflection [11] - Using the class Class we can gather information about join points. They contain information about classes, instance variables, methods and their slots. Using reflection at method call interception we can also gather information about the caller and the callee.
- Dynamic referencing of the super class - Almost all OO languages compile super as a static reference, which is calculated at compilation. The static approach can be problematic when implementing the calls to super classes inside the aspect.
- Slots – The Basic language COOL was extended in such a way that components could include explicit join points also called slots. Slots must also be accessible through reflection.

Using slots the programmer has the chance to explicitly define the possible places where the user can customize the component. We believe that slots as presented in [9] breaks the concept of encapsulation. In AspectCOOL we concentrated on achieving the highest possible freedom of customization while preserving encapsulation. Because we wanted to achieve separated compilation of such components, we also had to assure static safety. The idea was to offer the component developer a chance to specify the environment of the slot. With this approach developers can precisely define which variables can be used in slots. Since slots in COOL appear on the level of statements, we have to deal with three types of variables: instance variables, method arguments and method local variables. In the example below class in COOL with slots is presented.

```
class Account
{
  private balance: Integer;

  public create(aBalance:Integer): Account     // constructor
    balance = aBalance;
    return self;
```

```
  public deposit(aAmount:Integer) : Boolean
    balance = balance + aAmount;
    return true;

  public withdrawal(aAmount:Integer) : Boolean
    oldBalance :Integer=balance;
    result     :Boolean=true;
    if (aAmount>balance) then
      begin
        result = false;
        slot withdrawal_verify  (aAmount, balance, oldBalance, result);
      end
    else balance=balance-aAmount;
    return result;

  public getBalance(): Integer
    return balance;

  public getMaxWithdrawal():Integer
    return self.getBalance();

}
```

The class *Account* has four methods and one constructor. In the method *withdrawal* we have one slot which performs withdrawal verification. The class designer can specify the list of variables, which can be used in the slot. The slot *withdrawal_verify* uses four variables: *aAmount*, *balance*, *oldBalance* and *result*. The variable *aAmount* is method argument, *balance* is instance variable, *oldBalance* and *result* are local method variables. This slot also offers the component user the possibility to allow negative balance. In a case where withdrawal would bring account to a negative state, it is not automatically rejected. Using slot, a different withdrawal policy can be implemented. This means that a slot should use call by reference in order to perform these kind of task. In this example the slot would have set the *oldBalance*, subtract *aAmount* from the variable *balance* and modify the *result* variable to signal withdrawal success. These classes, which contain slots, are very similar to basic COOL classes. In case that user does not have the need to use slots, there is no difference in the way class is used. Slots, which are not used are ignored and therefore transparent for the user. Such classes can be compiled and distributed in object form to final users, which can be customized later without compilation.

3.2 Aspect language

The aspect part of the program consists of one or more aspect classes and the part of the program where aspects are applied on concrete join points. Therefore we divide the aspect program in two parts: aspect classes and the aspect application part.

3.2.1 Aspect classes

Each aspect class can define advices, which can be applied as actions on join points and slot advices which are applied on slots in component. Considering the time of application, we have four types of advices: before, enter, exit and after. Before and enter advices are actions executed before the execution of the actual method. Exit and after advices are executed after the execution of the method. The difference between the advices before and enter, and, the advices exit and after is in the execution environment. Before and after advices are executed in the callers environment while enter and exit advices are executed in environment of the executed method (Figure 1). Or simpler: the difference is in the class of self reference at the execution time of the advice. Compared to before and enter, the advices exit and after have the possibility to modify the value returned by the original method.

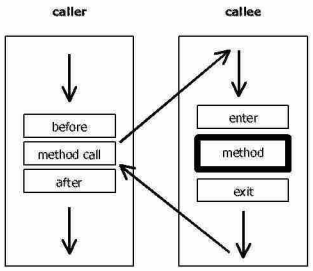

Figure 1: Advice types and environment of their execution.

The following example illustrates the definition of aspect class:

```
aspectclass AccountLimit
{
    log  :Log;
    negative_limit: Integer;

    create ()
      log = new Log();
      negative_limit=500;
    return;

    advice on Object log_call(value:Object)
      log.write(joinpoint.calleeMethod.getName()+" : "+value);
    return;

    advice on Account add_limit() : Integer
    return returnvalue + negative_limit;

    slot advice limit_implementation ( aAmount : Integer , balance : Integer,
                                oldBalance :Integer, result :Boolean)
      if (balance-aAmount > negative_limit) then
        begin
          oldBalance = balance;
          balance = balance - aAmount;
          result = true;
        end
      else result = false;
    return;
}
```

In the above example we have the *log* object to write the log file and Integer variable *negative_limit* to represent the negative limitation on the account. Aspect class can also have a constructor to perform the initialization of instance variables.

When an advice is declared, the class on which the advice can be applied (Account in add_limit advice), has to be provided. This is very important in the context of separate compilation because we have to perform a static analysis of the code in order to compile an advice. Advices can be applied on a variety of objects and this class name determines the type of self reference in an advice. The type or class of advice is considered when an advice is applied on concrete objects. At advice application it is also possible to use concept of inclusion polymorphism. Using this concept advice can be also applied on objects from subclasses. If needed, the programmer can also dynamically determine the class of object and perform suitable typecasting. Each advice has a list of formal arguments constructed from argument names and their types. Argument types are also necessary for static analysis of advices. These arguments are used for communication between an advice and the join points on which it is applied. The argument list can be followed by an advice return type. As mentioned before, we have four types of advices from which two (exit and after) can modify the return value. The advice return type specifies the class of value returned by the advice. This class can be,

considering inclusion polymorphism, the same class or a subclass of the class returned by the method on which the advice is applied. In the context of separate compilation, method calls in components are already compiled in such way that the caller expects the return value of some type. In case that type of return value is modified by an advice, it can only be modified to one of the subclasses of the original return type. Without this limitation we might come to the situation where the method on which an advice is applied, returns a value incompatible with the type expected by the method caller, resulting in runtime error.
Besides the arguments in the argument list, each advice gets additional implicit arguments (table 1). Before and enter advices get three arguments. The exit and after advice get an additional argument *returnvalue*:

Name	Type	Meaning
aspectself	same as class of aspect	reference on aspect instance
self	class on which advice can be applied (after »advice on«)	reference on object in which this advice executes
joinpoint	class JoinPoint	describes join point (see below)
returnvalue	return type of advice	used to modify methods return value

Table 1: Additional implicit arguments.

Class JoinPoint describes the join point on which advice was used. It has six fields:

Name	Type	Meaning
caller	Object	reference to object which called the method
callee	Object	reference to object whose method was called
callerMethod	Method	method from which original method was called
calleeMethod	Method	method which was called
callerParameters	Object[]	caller method parameter values
calleeParameters	Object[]	callee method parameter values

Table 2: Structure of JoinPoint.

Advice environment consists of aspect class instance variables, advice arguments and local advice variables. In our example we have two advices: *log_call* and *add_limit*. The advice *log_call* simply writes the log of the method call. It can be applied on any object in COOL. This advice writes the name of the called method and the argument value, which can be of any type, to the log file. The method name comes from the field *calleeMethod* in the JoinPoint class. The field *calleeMethod* is an object of the class Method that is used in reflection to describe methods.
The second advice named *add_limit* is intended to be applied either as an after or exit advice since it has a return type. This advice expects the methods to return an Integer value. The value returned by a method is modified by adding maximum allowed negative balance. Compared to *log_call* advice, this advice can be applied on instances of the class Account or its subclasses.
The advice class can also have one or more slot advices. A slot advice is described with a name, arguments and body. In our example one slot advice is declared. This slot advice has four arguments. Since slots represent explicit join points which can appear anywhere in the code on the statement level, there has to be communication between the slot and the surrounding statements. As we mentioned before, a slot, which would have access to the complete environment of surrounding statements, would break the principle of encapsulation. Therefore we chose the approach where the programmer exactly specifies a portion of the environment used by the slot. With this approach we transferred the decision about the content of the environment to the programmer. In our example, the slot advice *limit_implementation* implements the withdrawal policy, which allows negative balance to some predefined limit.

3.2.2 Aspect application part

After aspect classes have been written and compiled, they can be distributed in object form to the final users. These aspects can be later applied on their components. The application part of the program in AspectCOOL is the part where advices and slot advices are connected to concrete join points. In AspectCOOL we decided to describe join point as a method call from one method to another. This method call with corresponding methods is also called transition [12] and it supports the jumping aspect code [13]. It is obvious that in this case join point is specified with a pair of caller and callee methods. A method is described with a class, a method name, arguments and return type. Since advices are usually applied on a set of join points, a mechanism for the description of such a *transition set* is also needed. Wildcards can also be used for the description of a transition set. Besides the transition set, the aspect, which is applied, and its application time must be specified. For example:

```
on enter  *.*(..) : * -> Acc*.*(a: Object, ..) : * apply log_call(a)
```

describes the transition from any method to any method from class, whose name starts with »Acc«, and has the first argument of type Object. On this set of join points the advice »log_call« is applied. It is declared as »enter«, which means that it will be executed before the method call with self reference set to the receiver of method call. Method parameters from both transition methods can also be sent to the advice. In our example the first argument of the called method is of type Object, which is subsequently sent to advice »log_call« as an argument. It is important that arguments stated in the application also match the advice declaration. In the advice we can also get all other arguments using fields *callerParameters* and *calleeParameters* in the *joinpoint* argument. This is useful in cases when we specify very general join point sets covering very different combinations of parameters.

Similar to advices, slot advices also have to be applied on concrete component slots. A slot advice is applied in the following way:

```
on slot Account.withdrawal_verify apply limit_implementation;
```

A very important issue here is aspect management. The question is how many instances of aspect class are needed and which instance of aspect class is used on which join point. In AspectCOOL we use the following approach:

```
new AccountLimit {
  on enter  *.*(..) : * -> Acc*.*(a:Object, ..) : * apply log_call(a);
  on exit   *.*(..) : * -> Account.getMaxWithdrawal(..) : Integer apply add_limit();
  on slot Account.withdrawal_verify apply limit_implementation;
}
```

Using the operator new, an instance of the AccountLimit aspect class is created. The advices and slot advices used in the block correspond to that instance. It is also possible to specify that for each new instance of class a new instance of aspect class is created. In that case we have two possibilities. New instance of aspect class can be created for each callee or for each caller. For example:

```
new AccountLimit for each callee {
  on enter  *.*(..) : * -> Acc*.*(a:Object, ..) : * apply log_call(a);
}
```

would create a new instance of aspect class AccountLimit for each object (objects of classes, which name starts with »Acc«) whose method, which suits the method transition, is called.

4 Aspect Weaving

In this section the basic principle behind aspect weaving in AspectCOOL is presented. The basic idea used for aspect weaving is a method call interception. Using this concept a method call can be detected. At the moment when the method call is intercepted we have the possibility to perform additional operations before

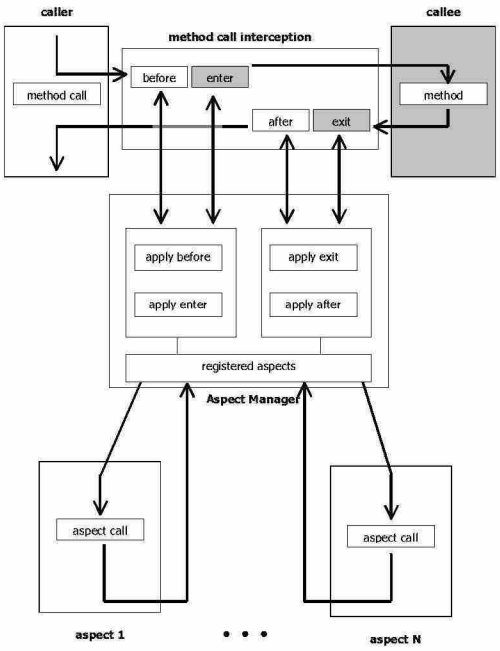

Figure 2: Aspect weaving in AspectCOOL.

and after the call of the original method. Because join points in object-oriented languages are represented by method calls we have the possibility for aspect weaving. In figure 2 the basic process of aspect weaving in AspectCOOL is presented. In the process of aspect weaving the main role is played by the Aspect Manager. The Aspect Manager is a component responsible for the registration of advices on join points. As presented in figure 2, when the method call takes place it is intercepted. The control flow is transferred from the caller to the method call interception. First, »before« advices are executed. Control flow is passed to the Aspect Manager, which then executes advices registered for that join point. After that, control is passed back to method call interception and the process is repeated for enter advices. Next, the method is executed followed by exit and after advices. After execution of each type of advice, the control flow is transferred from the aspect manager back to method call interception because the appropriate environment for each type of advice has to be assembled. For enter and exit advices the environment is bound to the callee object and for before and after advices to the caller object. For exit and after advices the return value also has to be added to the environment. The important issue here is also the optimization of the method call interception and advice execution. Since this approach of method interception for each method call introduces additional operations, this overhead has to be minimized as much as possible.

Similar to Aspect Manager, Slot Manager manages the registration of slot advices. Slot Manager is a component, which is used to apply slot advices to slots exported by components. In the component code, where a slot is exported as an explicit join point, a call to the Slot Manager is inserted. When the program control flow comes to the place where the slot is inserted, Slot Manager is called (figure 3). In case a slot advice is registered on that slot, a suitable environment is created and the control flow is passed to slot

advice. After slot advice has finished, parameter values are copied and control flow is returned back to the method.

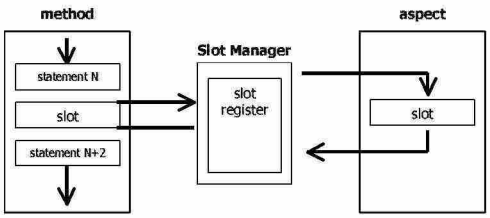

Figure 3: Slot Manager.

5 Conclusion

The idea of AOP was introduced in [1] and since then has gained in popularity. Many researchers [12], [14] are working on further developments of this idea. In order to investigate the usability and usefulness of AOP, different experiments were performed [15], which showed that AOP has a positive impact on separation of concerns.

AOP is currently supported mostly by aspect weavers, which require a source code for both components and aspects in order to create the final program. In this paper we have presented our approach to aspect weaving in order to achieve separate compilation. This approach to aspect weaving is used in the languages COOL and AspectCOOL, which are also presented in the paper. With this approach it is possible to apply aspects and slots on already compiled components. On the other hand this approach introduces additional overheads to programs. In further research we plan to do performance studies.

6 References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin. Aspect-Oriented Programming, in M. Aksit, S. Matsuoka (eds.) : ECOOP'97 – Object-Oriented Programming, Lecture Notes in Computer Science vol. 1241, pp. 220 - 242, Springer-Verlag, 1997.

[2] AspectJ homepage, www.aspectj.org

[3] M. Mernik, N. Korbar, V. Žumer. LISA: A Tool for Automatic Language Implementation, ACM SIGPLAN Notices, Vol. 30, No. 4, pp. 71 - 79, 1995.

[4] M. Mernik, V. Žumer, M. Lenič, E. Avdičaušević. Implementation of multiple attribute grammar inheritance in the tool LISA, ACM SIGPLAN Notices, Vol. 34, No. 6, pp. 68-75, 1999.

[5] C. V. Lopes. D: A Language Framework for Distributed Programming, Ph.D. Thesis. Graduate School of the College of Computer Science, Northeastern University, Boston, Massachusetts, 1997.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold. An Overview of AspectJ, ECOOP'01, Budapest, 2001.

[7] A. Beugnard. How to make aspects re-usable, a proposition, Position paper, Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99, 1999.

[8] O. Lehrmann Madsen. The Mjølner BETA Fragment System, in J. Lindskov Knudsen, M. Löfgren, O. Lehrmann Madsen, B. Magnusson (eds.): Object-Oriented Environments:The Mjølner Approach, Prentice Hall, 1994.

[9] J. Lindskov Knudsen. Aspect-Oriented Programming in BETA using Fragment System, Position paper at Workshop on Aspect-Oriented Programming, ECOOP'99, 1999.

[10] I. Welch, R. Stroud. Load-time Application of Aspects to Java COTS Software, Position paper, Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99, 1999.

[11] G. Kiczales, J. des Rivières, G. Bobrow. The Art of the Metaobject Protocol, MIT Press, 1991.

[12] R. Lämmel. Semantics of Aspect-Oriented Programming. Tehnical Report CWI, 2001.

[13] J. Brichau, W. de Meuter, K. de Volder. Jumping Aspects, Position paper at Workshop on Aspects and Dimensions of Concerns, ECOOP 2000, 2000.

[14] C. Constantinides, A. Bader, T. Elrad. An Aspect-Oriented Design Framework for Concurrent Systems, Position paper at Workshop on Aspect-Oriented Programming, ECOOP'99, 1999.

[15] R. J. Walker, E. L. A. Baniassad, G. C. Murphy. An Initial Assessment of Aspect-Oriented Programming, in Proceedings of the 21st International Conference on Software Engineering, pp. 120 – 130, ACM Press, 1999.